

Snakemake on Biowulf

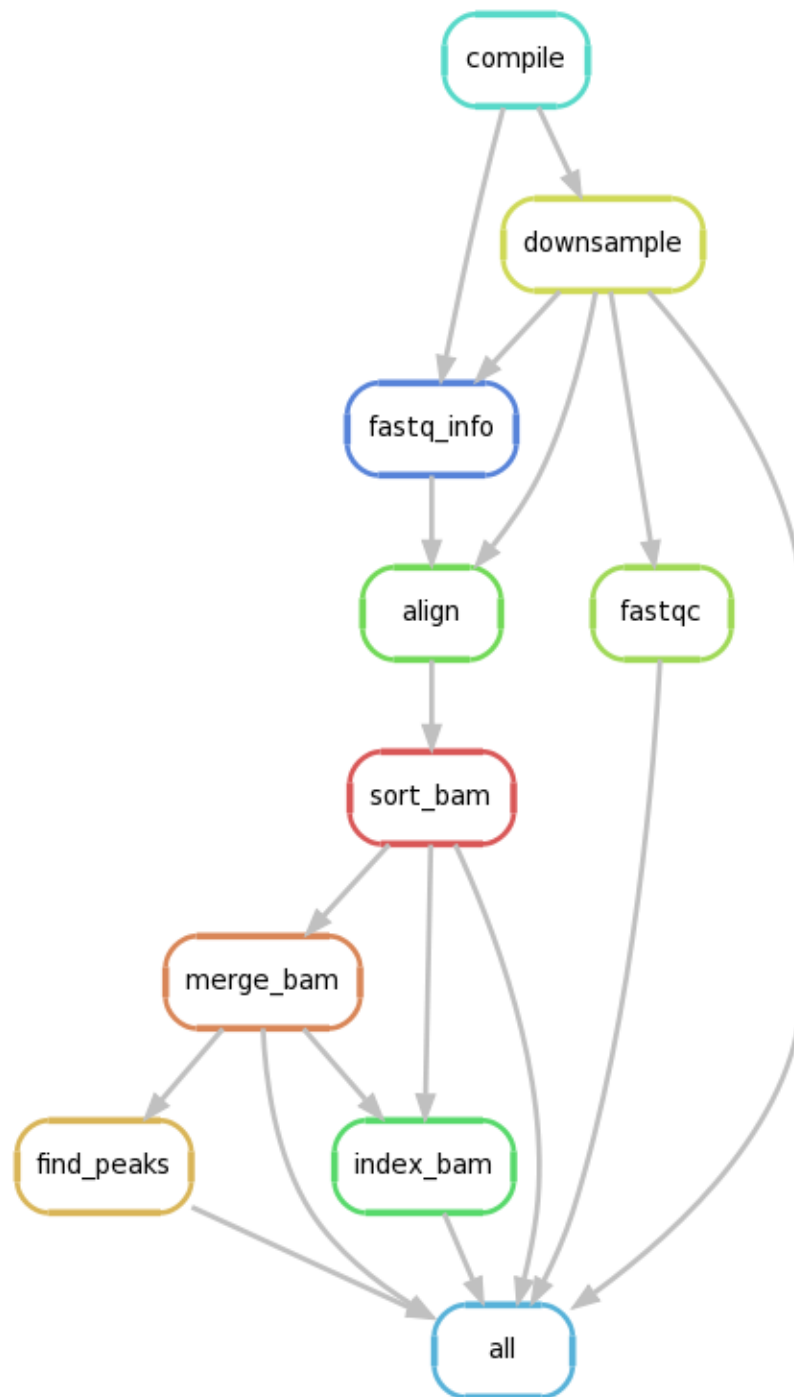
Example Project

A set of ChIP-Seq experiments of the same molecule in different genotypes.

Data arrives in chunks and needs to be cleaned, QCed, aligned.

Peaks need to be called and compared.

Tracks generated, data prepared for interactive analysis downstream, PCA, ...



We Would Like a Tool That

- can figure out how to run a whole workflow based on a set of rules for transforming one file type to another
- is reproducible
- reruns steps if necessary (input files change or processing steps change)
- runs any necessary steps automatically as new data is added

Snakemake Is a Rule Based Dependency Tracker

Rule

Rules describe how to transform one file type into another. Files are identified based on constant parts of their name (e.g. .fastq, _fastqc.zip, ...)

Dependency Tracker

Snakemake automatically determines what files are needed to produce a certain file type based on the rules. This information is used to calculate a dependency tree for the whole workflow. Rules are only executed if their outputs either don't exist or are older than the input files.

There Are Many Such Tools

make, ninja, scon, waf, ruffus, jug,
Rake, bpipe, paver, Galaxy, ...

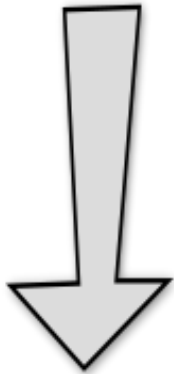
So Why Use Snakemake?

- Snakefiles are python code - i.e. a real programming language is available
- designed with bioinformatics in mind
- easy to offload processes to cluster nodes
- advanced pattern matching
- multiple input and output files
- many bonus features: configuration, wrappers, target lists, graphs of workflow, reports, ...
keeps track of code changes in rules

Rules

Rules describe how to transform one file type into another. Files are identified based on constant parts of their name (e.g. .fastq, _fastqc.zip, ...)

unsorted text file



sorted text file

```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```

Rules

```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```

Rules

Rules start with
the **rules** keyword



rule sort:

input: "words.txt"

output: "words.sorted.txt"

shell: "sort {input} > {output}"


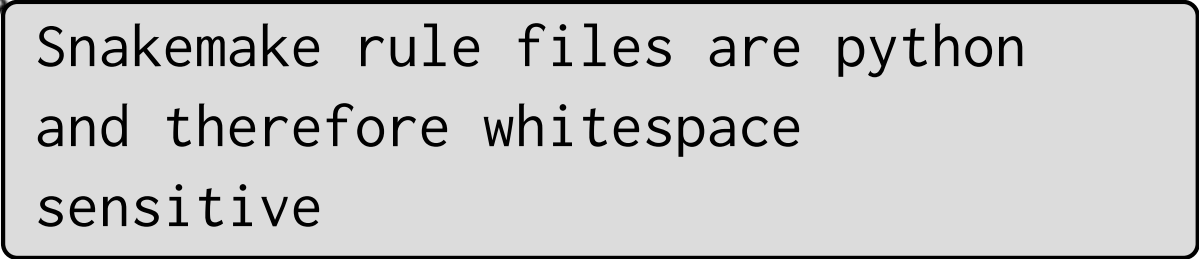
Rules

```
rule sort:
```

```
    input: "words.txt"
```

```
    output: "words.sorted.txt"
```

```
    shell: "sort {input} > {output}"
```



Snakemake rule files are python
and therefore whitespace
sensitive

Rules

Snakemake uses filenames
to determine which rules to
apply



```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```

Rules

```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```



Rules can use `shell:`,
python (`run:`), and R (`run: R()`)

Rules


```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```



Shell rules have to be quoted
with single or triple quotes

Rules

```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```



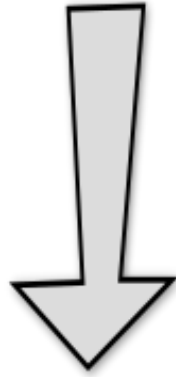
Use {...} to access input, output, parameters, threads, log, wildcards, and global variables. use {{...}} to get literal braces.

Rules

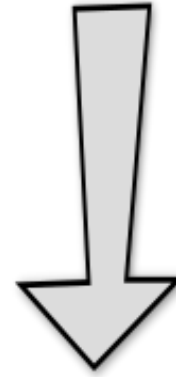
```
rule sort:  
  input: "words.txt"  
  output: "words.sorted.txt"  
  shell: "sort {input} > {output}"
```

Rules With Wildcards

unsorted text file unsorted text file



sorted text file



sorted text file

```
rule sort:
```

```
  input: "{name}.txt"
```

```
  output: "{name}.sorted.txt"
```

```
  shell: "sort {input} > {output}"
```

Live Demo

`/data/classes/snakemake/01_sort_text_files`

```
cp -r /data/classes/snakemake/01_sort_text_files .  
cd 01_sort_text_files  
ls -lh
```

total 16K

-rw-rw----	1	wresch	staff	218	Mar	28	19:20	numbers1.txt
-rw-rw----	1	wresch	staff	217	Mar	28	17:43	numbers2.txt
-rw-rw----	1	wresch	staff	2.9K	Mar	28	19:58	README.md
-rw-rw----	1	wresch	staff	506	Mar	28	23:02	Snakefile

cat Snakefile

```
localrules: all, clean, merge
```

```
rule all:
```

```
    input: "numbers1-numbers2.txt"
```

```
rule merge:
```

```
    input: "{name1}.sorted.txt",  
           "{name2}.sorted.txt"
```

```
    output: "{name1}-{name2}.txt"
```

```
    shell: "join -j1 -a1 -a2 -o 0 1.2 2.2 -eND {input} > {output}"
```

```
rule sort:
```

```
    input: "{name}.txt"
```

```
    output: "{name}.sorted.txt"
```

```
    log: "sort-%j.out"
```

```
    shell: "sort -k1,1 {input} > {output}"
```

```
rule clean:
```

```
    shell: "rm -f *.sorted.txt numbers1-numbers2.txt"
```

```
$ module load snakemake
# or
$ module load python/3.4.3
```

```
$ snakemake -s Snakefile all
# since snakemake defaults to executing the first target in the
# rule file 'Snakefile' this is equivalent to
$ snakemake
```

Provided cores: 1

Rules claiming more threads will be scaled down.

Job counts:

count	jobs
1	all
1	merge
2	sort
4	

rule sort:

input: numbers1.txt

output: numbers1.sorted.txt

Rerunnig snakemake if nothing changed is a no-op

```
$ snakemake
```

```
Nothing to be done.
```


Clean up

```
$ snakemake clean
```

-n, --dry-run: run without executing rules
-p: print shell commands

```
$ snakemake -np
```

```
rule sort:
```

```
    input: numbers2.txt
```

```
    output: numbers2.sorted.txt
```

```
    log: sort-%j.out
```

```
sort -k1,1 numbers2.txt > numbers2.sorted.txt
```

```
[...snip...]
```

--verbose: more information
--reason: show reason why rule is being
executed

```
$ snakemake -n --reason --verbose
```

```
Resources before job selection: {'_cores': 1, '_nodes': 0}
```

```
Ready jobs (2):
```

```
    sort
```

```
    sort
```

```
Selected jobs (1):
```

```
    sort
```

```
Resources after job selection: {'_cores': 0, '_nodes': 0}
```

```
rule sort:
```

```
    input: numbers2.txt
```

```
    output: numbers2.sorted.txt
```

```
    log: sort-%j.out
```

```
    reason: Missing output files: numbers2.sorted.txt
```

```
[...snip...]
```

--list, -l: list all rules

```
$ snakemake --list
```

```
all  
merge  
sort  
clean
```

Use -S, --summary to show tabular information about all files generated by the workflow.

```
$ snakemake --summary
```

output_file	date	rule	version	status	plan
numbers1-numbers2.txt	-	merge	-	missing	update pending
numbers2.sorted.txt	-	sort	-	missing	update pending
numbers1.sorted.txt	-	sort	-	missing	update pending

```
$ snakemake
```

```
$ snakemake --summary
```

output_file	date	rule	version	status	plan
numbers1-numbers2.txt	Mar 29 2016	merge	-	ok	no update
numbers2.sorted.txt	Mar 29 2016	sort	-	ok	no update
numbers1.sorted.txt	Mar 29 2016	sort	-	ok	no update

```
$ snakemake
```

```
Nothing to be done.
```

```
$ touch numbers1.txt
```

```
$ snakemake --summary
```

output_file	date	rule	version	status	plan
numbers1-numbers2.txt	Mar 29 2016	merge	-	ok	update pending
numbers2.sorted.txt	Mar 29 2016	sort	-	ok	no update
numbers1.sorted.txt	Mar 29 2016	sort	-	updated input files	update pending

```
$ snakemake -r
```

```
[...snip...]
```

```
rule sort:
```

```
    input: numbers1.txt
```

```
    output: numbers1.sorted.txt
```

```
    log: sort-%j.out
```

```
    reason: Updated input files: numbers1.txt
```

What happens when the code executed by a rule changes? Edit a rule and run

```
$ snakemake
```

```
Nothing to be done.
```

Targets are not automatically recreated but code changes are tracked:

```
$ snakemake --summary
```

output_file	date	rule	version	status	plan
numbers1-numbers2.txt	Mar 29 2016	merge	-	rule implementation changed	no update
numbers2.sorted.txt	Mar 29 2016	sort	-	ok	no update
numbers1.sorted.txt	Mar 29 2016	sort	-	ok	no update

```
$ snakemake --lc # or --list-code-changes
```

```
numbers1-numbers2.txt
```

Force recreation of all affected targets with

```
$ snakemake -R $(snakemake --lc)
```

Rules can be given explicit versions

```
rule merge:
    input: "{name1}.sorted.txt",
           "{name2}.sorted.txt"
    output: "{name1}-{name2}.txt"
    version: "1.0"
    shell: "join -j1 -a1 -a2 -o 0 1.2 2.2 -eND {input} > {output}"
```

```
$ snakemake --lv # or --list-version-changes
$ snakemake -R $(snakemake --lv)
```


Param changes are tracked as well

```
$ snakemake --lp # or --list-param-changes  
$ snakemake -R $(snakemake --lp)
```

Running in Parallel - Locally

Snakemake workflows can be run in parallel on the local machine. `-j` specifies the number of cpus snakemake is allowed to use.

Snakemake will use the information from the `threads` section to determine how many jobs can be run at the same time.

Don't do this on the login node

Running in Parallel - On Cluster

Snakemake workflows can be run in parallel on the biowulf cluster. `-j` specifies the number of jobs to run concurrently.

Limit local CPUs with `--local-cores`

Please run the master process as a batch job or from an interactive session.

Submission is done via a template string provided with `--cluster`

```
$ snakemake --cluster "sbatch --time=5 --mem=50m --partition=quick"  
$ snakemake --cluster "sbatch --cpus-per-task={threads}"  
$ snakemake --cluster "sbatch --mem={params.mem}"
```